Editorial Manager(tm) for Journal of Physics: Conference Series
Manuscript Draft

Manuscript Number:

Title: The JANA calibrations and conditions database API

Article Type: Contributed

Corresponding Author: Dr. David Lawrence, Ph.D.

Corresponding Author's Institution: Jefferson Lab

First Author: David Lawrence, Ph.D.

Order of Authors: David Lawrence, Ph.D.

The JANA calibrations and conditions database API

David Lawrence
12000 Jefferson Ave. Suite 8; Newport News, VA 23601; USA
E-mail: davidl@jlab.org

Abstract. Calibrations and conditions databases can be accessed from within the JANA Event Processing framework through the API defined in its JCalibration base class. The API is designed to support everything from databases, to web services to flat files for the backend. A Web Service backend using the gSOAP toolkit has been implemented which is particularly interesting since it addresses many modern cybersecurity issues including support for SSL. The API allows constants to be retrieved through a single line of C++ code with most of the context, including the transport mechanism, being implied by the run currently being analyzed and the environment relieving developers from implementing such details.

1. Introduction

The JANA[1] event processing framework is being developed primarily for the GlueX experiment at Jefferson Lab in Newport News, Virginia, USA. JANA is a C++, multi-threaded event processing framework. One of the important features it provides is a common mechanism that can be used by all processing threads to access the Calibrations and Conditions Database (CCDB). Though the framework is motivated by the GlueX experiment, it is designed to be of general use. This generalization has an advantage in that it provides a well defined application programming interface (API) (see figure 1) which allows quicker startup of code development as described in the following section.

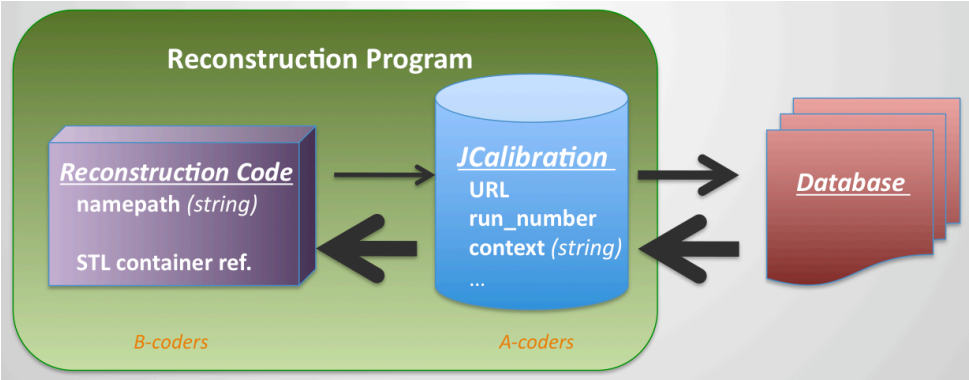

Figure 1. The *JCalibration* class implements the API and serves as a well-defined boundary between the reconstruction code and the database itself.

2. Providing a Framework for Development

In the earlier stages of development of the simulation and reconstruction source code, it is important to provide a framework that can be used by collaborators to begin development quickly, yet allow the code to be useful throughout the lifetime of the experiment. Naturally, not all code developed early on will survive through the whole experiment. However, common experience shows there are always a few pieces of code that were thought to be temporary when first written, but become an integral part of the final package. Developing against a well defined API helps ensure a level of modularity that will ultimately make the legacy code more maintainable.

Developing all features of the framework prior to starting work on the reconstruction code is not practical. In the case of the CCDB, for example, one does not develop the entire database system first before beginning to develop the simulation or reconstruction software. However, one can define an API that would be used by the bulk of simulation/reconstruction software and a simple back end that would then allow development to begin with relatively little effort. The full CCDB design is then deferred until later or done in parallel to the development of the other software pieces. In this spirit, JANA implements a trivial CCDB backend based on ASCII files that can be used for initial development. The API is implemented through the *JCalibration* base class which all backend implementations must inherit from. The trivial backend is implemented in the *JCalibrationFile* class and can be used in lieu of a full database while supporting the API that will be used to interface with the eventual CCDB.

2.1. Generator Classes and Discovery Mechanism

Supporting a simple backend for development and later, a full featured database will lead to a transition period during which it is desirable to have both methods accessible in the same executable. Supporting multiple backends simultaneously is done easily through a traditional object oriented factory mechanism. To supply a backend, one needs to supply 2 classes: 1.) the class implementing the backend itself (*e.g. JCalibrationMySQL*), and 2.) a generator class (*e.g. JCalibrationGeneratorMySQL*). The generator class is used to:

(i) Determine which backend is appropriate for a specified CCDB

(ii) Create an object of the appropriate *JCalibration*-based class

Figure 2 lists the methods that must be supplied by a generator class (i.e. one inheriting from *JCalibrationGenerator*). Two of the three methods (*Description()* and *MakeJCalibration()*) are trivial. The *CheckOpenable()* method returns a floating point value between 0 and 1 indicating the probability that it can open and read from the specified url, run, and context. The *url* is a free-form string indicating the protocol and location of the CCDB data. For example, this could have a form:

file:///group/halld/calib

or

mysql://halld_user@halld_db.jlab.org

The *run* value is the run number and is assumed to be the primary index of the CCDB. The *context* parameter is a free-form string that can be used to convey additional information about the specific constant set desired. For example, the string "datetime=2009-05-14-6-44;tag=davetest" could be used to request the constants tagged "davetest" that would have been returned had the query been made on May 14th, 2009 at 6:44AM. The exact format of the *context* string and what settings it supports (if any) is determined by the backend.

```
virtual const char* Description(void)
virtual double CheckOpenable(string url, int run, string context)
virtual JCalibration* MakeJCalibration(string url, int run, string context)
```

Figure 2. Purely virtual methods that must be supplied when inheriting from the *JCalibrationGenerator* class.

3. A Simple API

The JANA CCDB API is designed to provide as simple an interface as possible to the simulation/reconstruction code author. Most information about the specifics of the particular data set and its location are hidden in the sense that they need not be explicitly provided whenever a particular item is retrieved. The underlying principle being that the location, protocol , and particular constant set desired is the same for all packages processing that event. The implied context (location, protocol, run number, ...) in which the request for an item[1] is made is kept in a common location, the *JCalibration* object.

Figure 3 shows the core methods of the API. These are templates so that the exact data type is determined by the caller. Types are converted to the specified type using the ANSI stringstream class. Conversion from strings is assumed to be necessary at some point since most common database systems transport data as strings to avoid byte-ordering or word length issues. Therefore, if a reference to a container of type $vector < double >$ is passed, then the values will be retrieved from the backend as strings and the stringstream class used to convert them to type *double* as they are copied into the container.

Not shown in figure 3 is another set of four *Get()* methods that are identical to the ones shown except references to *pointers* to *const* containers are used. These work in a similar way except they indicate that the *JCalibration* class should create a container of the specified type and only return a const pointer to it. Then, for subsequent requests for the same constants (e.g. by other threads) the same const pointer will be returned such that only one copy of the constants will be kept in memory at a time.

In JANA reconstruction algorithms are implemented in "factory" classes. A factory class implements a set of callback methods that the framework calls during the course of event processing. One of these is the *brun()* method which is called whenever a run number changes is sensed. This is primarily used to indicate to a factory that the calibration may have changed since the run number is used as the primary index. Figure 4 shows an example of how the API is used to retrieve a set of 3 parameters, indexed by name. In this example, the values are kept as local data members in the factory class to be used by the specific algorithm it implements. The first parameter is called the *namepath*. This free-form string is passed to the backend which must then parse it to identify the specific item that is desired. The suggested convention is to use a hierarchical type syntax with the first part indicating the detector system to which the item belongs. This allows more complicated systems to implement a "deeper" hierarchy if needed.

Notice that in figure 4 the call is actually made to the *GetCalib()* method of the JANA framework's *JEventLoop* class. This is so the framework can identify the specific *JCalibration* object which it should route the call to. Since a multi-threaded program may be processing events from several runs simultaneously, several *JCalibration* objects may exist.

Figure 5 gives an example of how to access constant indexed by position. This example shows how an explicit error check may be made to verify that the correct number of values was obtained from the CCDB. The API itself does not supply mechanisms to check the number

[1] Here, "item" means either a 1-D array or a 2-D table.

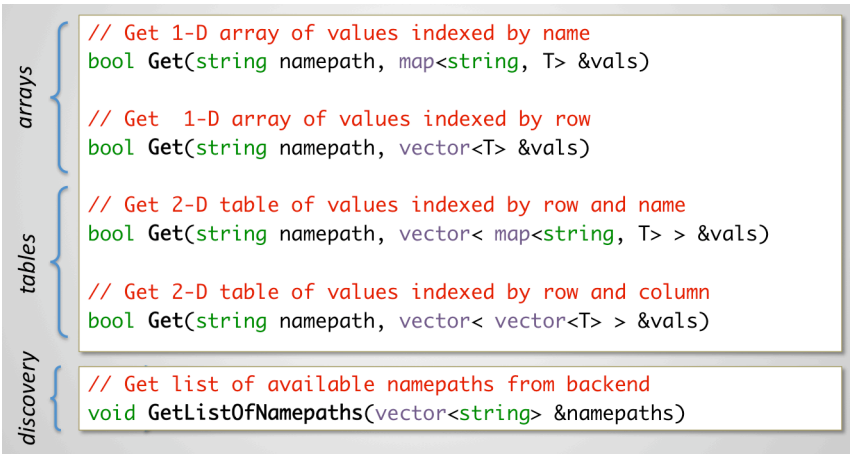

Figure 3. Templated methods that form the core of the API as seen by the user code. Constants may be retrieved as 1-D arrays or 2-D tables and may be indexed either by name (string) or position (number). The *GetListOfNamepaths()* method provides a minimal discovery mechanism allowing the list of available namepaths to be obtained from the backend.

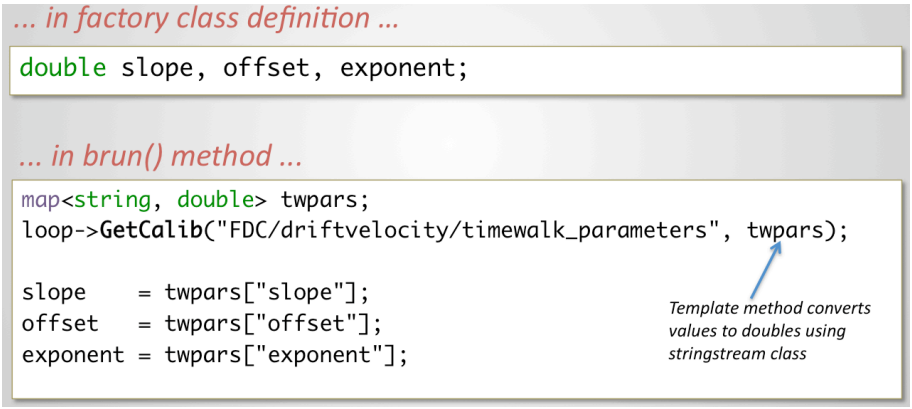

Figure 4. Example showing how to obtain constants indexed by name using an STL map container. In this example, the constants are retrieved(see text) whenever the run number changes and kept in local data members of the algorithm object.

or type of values supplied by the CCDB. Leaving this to the caller maintains a lower level of simplicity for the API which in turn, provides a higher level of flexibility in the CCDB design.

Figure 6 shows another example of obtaining values, but where the memory used to store them is owned by the JANA framework. This allows multiple consumers of the same set of constants to refer to the same set of values in memory, reducing the overall memory needed.

4. Database Backends

Figure 7 shows the 3 API methods that must be supplied in order to create a CCDB backend. Owing to how most databases transport using strings, only string types are used. Values in the CCDB backend are expected to be indexed by either name (e.g. figure 4) or position (e.g. figure 5). The backend is responsible for providing the indexes as strings so if only position indexing is available, the strings "0", "1", "2", ... will be used guaranteeing that order will be maintained

```
... in factory class definition ...
vector<double> tof_tdc_offsets;


... in brun() method ...
loop->GetCalib("TOF/tdc_offsets", tof_tdc_offsets);
if(tof_tdc_offsets.size()!=Ntof) throw JException("Bad Ntof!");


... in evnt() method ...
 double t = tof->tdc - tof_tdc_offsets[tof->id];
```

Figure 5. Example showing how to obtain constants indexed by position using an STL vector container. In this example, the constants are retrieved(see text) whenever the run number changes and kept in local data members of the algorithm object.

```
... in factory class definition ...
 const double *fcal_gains;


... in brun() method ...
const vector<double> *my_fcal_gains;
loop->GetCalib("FCAL/Energy/gains", my_fcal_gains);
fcal_gains = &(my_fcal_gains->front());


  ... in evnt() method ...
double Ecorr = fcal_hit->E * fcal_gains[fcal_hit->id];

fcal_gains[3] =1.2; // This will generate compile time error!
```

Figure 6. Example code showing how to get access to constants while allowing the *JCalibration* object to retain ownership of the constants. This reduces memory usage since multiple threads will refer back to the same location for the constants.

```
virtual bool GetCalib(string namepath, map<string, string> &svals)
virtual bool GetCalib(string namepath, vector< map<string, string> > &svals)
virtual void GetListOfNamepaths(vector<string> &namepaths)
```

Figure 7. API for the Calibrations and Conditions Database backend. Backends need only to supply these 3 methods in the *JCalibration*-derived base class. Strings are used since most databases communicate using strings to avoid byte ordering and word length issues.

with the map container[2].

[2] STL map containers will order the entries by hashing the key. This leads to alphabetical order for words, but numerical ordering for numbers.

4.1. SOAP-based Web Service

The SOAP[2] standard is a commonly used mechanism by which data objects can be transported across the network, allowing one to develop distributed applications. It is a service oriented architecture (SOA) where a server publishes its methods and the definitions of the data objects used to pass information in and out of them in the form of a downloadable XML file (WSDL). The protocol works over HTTP (port 80) or HTTPS (port 443) with the latter providing an encrypted connection through a secure socket layer (SSL). SOAP is attractive as a communication mechanism because it works over HTTP(S) which bypasses the need to open additional ports in a firewall for remote access to a database.

JANA comes with a working example of a *JCalibration* backend that uses SOAP to obtain calibration constants over the network. The example uses the C++ SOAP implementation, gSOAP[3]. The gSOAP package fully supports SSL using the openssl[4] library. Thus, it is able to communicate using an encrypted connection with both server-side and client-side authentication via certificates. This is useful in restricting access to the CCDB to collaboration members while still allowing remote access from anywhere on the internet.

Because the communication is done as a web service, the actual database communication may be done via any programming language (e.g. Java, python, ...). A C++ implementation using gSOAP that can be run via CGI is included with the JANA source code. WIth this example, one can easily adapt any backend as a web service thereby allowing development of the full CCDB on a local network before publishing it as a Web Service.

5. Summary

The JANA Calibrations and Conditions Database API provides a simple interface for accessing constants. The well-defined API allows new projects to defer development or implementation of the full database and immediately start on development of simulation and reconstruction code without the need to rewrite it later once the full database is in place.

Working examples of using the C++ gSOAP package to access constants through an SSL encrypted connection are included in the JANA source code.

References

[1] D Lawrence. Multi-threaded event reconstruction with jana. *Journal of Physics: Conference Series*, 119(4):042018 (6pp), 2008.

[2] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. Technical Report NOTE-SOAP-20000508, W3.org, http://www.w3.org/TR/2000/NOTE-SOAP-20000508, July 2003.

[3] CCGrid2002, editor. *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks*. 2nd IEEE International Symposium on Cluster Computing and the Grid, May 2002.

[4] http://www.openssl.org/.